

# HERANÇA

**HERANÇA SIMPLES (cf. JAVA):**

**QUALQUER CLASSE POSSUI UMA E UMA SÓ SUPERCLASSE; A CLASSE OBJECT NÃO TEM SUPERCLASSE, É O TOPO DA HIERARQUIA.**

**QUESTÕES SOBRE HERANÇA:**

- 1) JAVA PERMITE QUE UMA SUBCLASSE POSSA REDECLARAR UMA VARIÁVEL DE INSTÂNCIA HERDADA E REDEFINIR O CÓDIGO DE UM MÉTODO DE INSTÂNCIA ?**

**Exº:**

```
public class A {  
    int i = 0;  
    int m() { return i; }  
}
```

```
public class B extends A {  
    int i = 1; // i de A é "shadowed"  
    int m() { return i; } // m é "overriden"  
}
```

**RESPOSTA: SIM !!**

---

**2) É POSSÍVEL, NO CONTEXTO DA CLASSE B, TER ACESSO QUER À DEFINIÇÕES LOCAIS QUER ÀS DE A QUE FORAM REDEFINIDAS ? SE SIM, COMO SE FAZ ?**

**RESPOSTA: SIM !!**

Por exemplo, um método de B poderia ter o seguinte código:

```
public int mb() {  
    i = super.i + 10;  
    return i + this.m() + super.m();  
}
```

**i** -> local  
**super.i** -> i da superclasse  
**this.m()** -> local  
**super.m()** -> m() da superclasse

**super** -> variável especial que refere o contexto da superclasse;

---

---

**3) TUDO PODE SER REDECLARADO E REDEFINIDO NUMA SUBCLASSE ?**

**RESPOSTA: NÃO !!**

Variáveis e métodos **final** não podem;  
Métodos **static** (de classe) não podem;  
Variáveis e métodos **private** não podem.

---

---

**4) QUE VARIÁVEIS E MÉTODOS (DE CLASSE E DE INSTÂNCIA) SÃO AUTOMATICAMENTE HERDADOS POR UMA SUBCLASSE ?**

**RESPOSTA:** De instância todos os que não forem **private** ! De classe nenhum, mas variáveis de classe podem ser redefinidas. Os construtores também não são herdados.

---

---

**5) OS MODIFICADORES DE ACESSO TÊM INFLUÊNCIA NA HERANÇA ?**

**RESPOSTA:** Os **private** não são herdados. Os outros são todos herdados e apenas definem quem lhes pode aceder, cf.

<b>private</b>	Apenas acessíveis ao código na classe que os declara;
<b>public</b>	Acessível a qualquer classe de qualquer package;
<b>protected</b>	Acessível a qualquer classe do mesmo package <b>E</b> ainda às suas subclasses mesmo que de outro package;
<b>"package"</b>	Acessível a qualquer classe do mesmo package;

---

---

## 6) HÁ COMPATIBILIDADE ENTRE UMA CLASSE E AS SUAS SUBCLASSES ?

**RESPOSTA:** Sim, mas apenas num sentido, ou seja, se **B** e **C** são subclasses de **A**, então é possível atribuir a uma variável declarada como sendo do tipo **A** uma instância de qualquer das suas subclasses, seja **B** ou **C**.

Exº:

```
A a1;  
B b1 = new B(); C c1 = new C();  
.....  
a1 = b1; // OK !!  
.....  
a1 = c1; // OK !!  
  
A a2 = new B(); // OK !!  
A a3 = new C(); // OK !!  
B b2 = new C(); // KO !!
```

---

---

## 7) MAS SE HÁ COMPATIBILIDADE ENTRE UMA CLASSE E AS SUAS SUBCLASSES, NUNCA SE PODE SABER AO CERTO QUAL A INSTÂNCIA CONTIDA NUMA VARIÁVEL DE TIPO **A** SE A CLASSE **A** TIVER VÁRIAS SUBCLASSES ? E ISSO É BOM OU É MAU ?

**RESPOSTA:** Correcto. Durante a compilação o **compilador** verifica o tipo declarado da variável e apenas valida se é uma instância

dessa classe ou de uma subclasse que lhe é atribuída.

Em tempo de execução, só o **interpretador** pode determinar qual a classe efectiva da instância contida numa variável de tipo **A**, e assim usar os métodos correctos dessa tal classe que pode portanto não ser **A**.

Esta possibilidade, designada **polimorfismo**, é uma das características fundamentais das linguagens de PPO. É fundamental para a programação genérica e extensível. Seja a declaração:

```
A a1 = new B();
```

Cada variável tem portanto um **tipo estático** (o que é determinado pelo compilador, cf. **a1** é do tipo estático **A**) e tem ainda um **tipo dinâmico** (determinado pelo interpretador ao analisar o conteúdo dinâmico de **a1**, sendo neste caso do tipo **B** após a atribuição **new B()**).

Para tal o interpretador usa um método designado **getClass()** que determina a classe da instância contida numa dada variável; o método **getName()** devolve a String que é o nome dessa classe, cf.

```
a1.getClass().getName() -> "B"
```

Veremos em seguida as enormes vantagens de possuímos este tipo de **polimorfismo** numa linguagem de programação.

---

**8) CONSIDEREM-SE AS DUAS CLASSES A E B APRESENTADAS NO INÍCIO, E CONSIDERE-SE O SEGUINTE PROGRAMA DE TESTE:**

```
public class A {  
    int i = 0;  
    int m() { return i; }  
}
```

```
public class B extends A {  
    int i = 1; // i de A é "shadowed"  
    int m() { return i; } // m é "overriden"  
}
```

```
public static void main(String[] args) {  
    A a = new B();  
    out.println(a.m());  
}
```

**QUAL O RESULTADO DO PROGRAMA E PORQUÊ?**

**RESPOSTA:** O resultado é, de facto, **1** e não **0**. **JAVA** funciona seguindo duas fases distintas fundamentais:

- a) **Compilação;**
- b) **Execução por interpretação;**

**COMPILAÇÃO:** Sintaxe correcta?

**EXECUÇÃO:** Que método executar de facto?  
Em que contexto?

No exemplo, o **compilador** verifica o tipo declarado da variável e valida a atribuição pois **B é subclasse de A**. Valida também a mensagem **m()** enviada a **a**, pois qualquer que seja a instância contida em **a**, a classe **A** pode sempre executar **m()**. Caso **m()** não esteja redefinido em **B** (não exista na subclasse) é herdado de **A!!** Assim, TUDO OK no **compilador**. Em tempo de execução o problema do interpretador é outro. O interpretador tem que determinar que método **m()** vai executar e há muitas hipóteses; o método **m()** de **A** se em **a** estiver uma instância de **A**; se em **a** estiver uma instância de uma subclasse de **A**, então o interpretador deverá executar ESSE método **m()** da subclasse, se existir, ou o de **A** caso não exista, pois é herdado.

No exemplo, a classe **B** redefine **m()** e o seu método **m()** ao ser executado dará como resultado **1**.

Assim, é em tempo de execução que o **interpretador**, em função do **tipo dinâmico da variável** decide qual o **método correcto** a ser enviado à instância contida na variável (e executado). Este mecanismo excepcional designa-se por tal razão **dynamic binding** (**associação dinâmica**) e **dynamic method lookup** (**procura dinâmica de métodos**).

Se a classe **A** tiver várias subclasses, o **interpretador** garante com este mecanismo que invocará o método **m()** da subclasse correcta, em função do tipo da instância contida na variável, que, por ser da **superclasse**, serve de receptora da mensagem tornando compatível tal código com todas as subclasses, sem ter que as distinguir através de um "case" como na programação imperativa.

O método **m()**, na pior das hipóteses, não existe em nenhuma subclasse, sendo herdado de **A**. Caso exista em todas com a sua adequada definição, será sempre executado o **mais próximo da subclasse**, ou seja, **o mais baixo na hierarquia**.

---

**9) CERTO! MAS TEM DE FACTO ALGUMA VANTAGEM QUE COMPENSE ESTAS COMPLICAÇÕES DE TIPOS ?**

**RESPOSTA:** Tem uma vantagem inigualável em qualquer outro paradigma: A facilidade de **GENERALIZAÇÃO DO CÓDIGO !!**  
**(1 linha vale por 1 milhão agora e no futuro)**

Nada melhor do que ver tais vantagens usando um exemplo com código concreto. Uma classe **Forma**, sem grande sentido, que serve apenas como **supertipo/superclasse** de **Circulo** e de **Quadrado**. Método comum para teste: **area()**.



**Exº:**

```
public class Forma {  
    public double area() { return 0.0; }  
}
```

```
public class Circulo extends Forma {  
    .....  
    public double area() {  
        return PI*pow(raio,2);  
    }  
}
```

```
public class Quadrado extends Forma {  
    .....  
    public double area() {  
        return lado*lado;  
    }  
}
```

**Vamos criar uma array contendo diversas formas, ou seja, instâncias de Forma, de Circulo e de Quadrado, sem sabermos mesmo onde as colocamos no array, cf.**

```
Forma[] formas = new Forma[50];  
formas[0] = new Forma(); .....  
formas[i] = new ???();
```

```
// Determinação da área de cada forma  
// indicando até o seu tipo ou seja a sua classe  
for(Forma f : formas) {  
    out.printf("Sou %s&n", f.getClass().getName());  
    out.printf("Area = %6.2f&n", f.area());  
}
```

```
// Somatório das áreas
double soma = 0.0;
for(Forma f : formas) soma += f.area();
out.printf("Area Total = %6.2f%n", soma);
}
```

Vejamos o que acontece quando, posteriormente, se tem que acrescentar uma nova subclasse:

```
public class Triangulo extends Forma {
    .....
    public double area() { return base*altura; }
}
```

**PERGUNTA:** O que temos que alterar no código anterior para agora lidarmos com triângulos ?

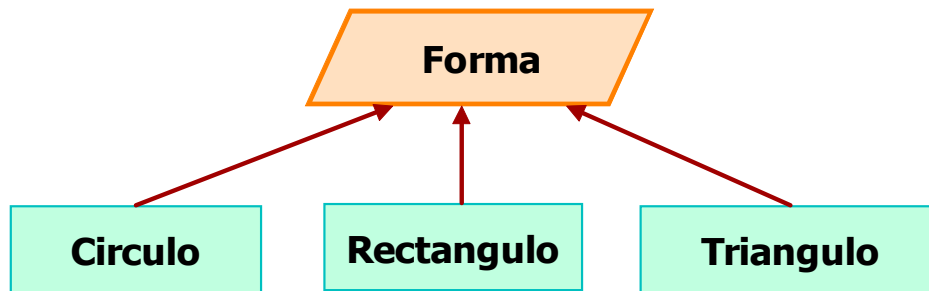
**RESPOSTA: NADA !!! ->>>>> EXTENSIBILIDADE**

---

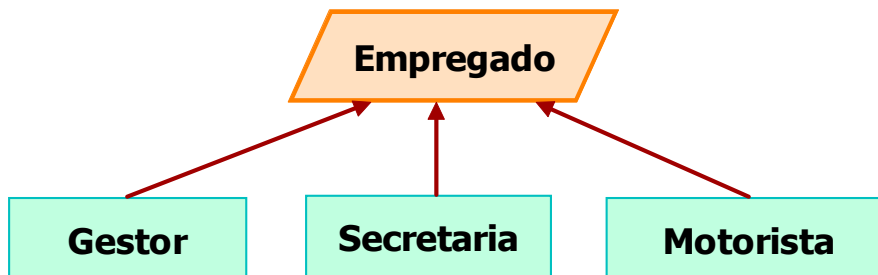
**NOTA FINAL:** A estratégia para uma boa utilização do **polimorfismo** e do mecanismo de **dynamic binding** é, sempre que tivermos diferentes classes com idêntico comportamento, **criarmos uma superclasse** destas e depois **usarmos apenas variáveis do tipo da superclasse para conter instâncias de qualquer subclasse**. O supertipo será compatível com as subclasses, sendo sempre o interpretador a decidir de forma automática, em "run time", qual o método correcto a utilizar em função da instância contida em tal variável.

---

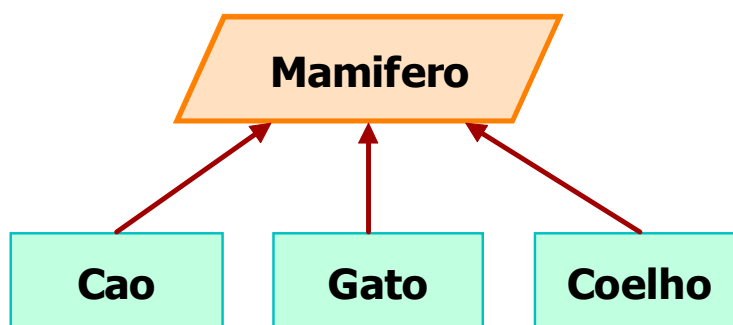
## HIERARQUIAS TÍPICAS:



**area()**  
**perímetro()**  
**numLados()**



**vencimento()**  
**idade()**  
**numFaltas()**



**peso()**  
**família()**  
**raçaDia()**